VŠB – Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Feature Description and Detection with Deep Learning

Detekce a deskripce příznaků pomocí hlubokého učení

2020 Vojtěch Ihn

VŠB - Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Diploma Thesis Assignment

Student: **Bc. Vojtěch Ihn**

Study Programme: N2647 Information and Communication Technology

Study Branch: 2612T025 Computer Science and Technology

Title: Feature Description and Detection with Deep Learning
Detekce a deskripce příznaků pomocí hlubokého učení

The thesis language: English

Description:

The aim of this diploma thesis is to explore the possibilities of deep learning techniques in the area of key point detection and feature description. The resulting application or library should be able to provide same functionality as the traditional keypoint detection and description frameworks (e.g. SIFT). In addition, applied algorithms should be based on machine learning methods for each part of the standard pipeline for local feature detection and description. The recommended framework is TensorFlow or Darknet with OpenCV. C/C++ or Python is required as a programming language. Clean and readable code in case of own implementation or thorough investigation and experimentation of already implemented algorithms is expected.

1. Investigate the area of key point detection and feature description in the context of deep learning techniques.
2. Devise an approach in which you will clearly describe your outcomes and desired objectives for individual parts of the aforementioned pipeline (i.e. key point detection, orientation estimation, feature description, feature matching).
3. Collect samples for training and testing of the individual stages of your approach.
4. Implement the proposed method in parts and perform thorough analysis.
5. Evaluate the overall performance of the developed pipeline.
6. Carefully document your entire workflow.

References:

[1] Ono, Yuki, Eduard Trulls, Pascal Fua, and Kwang Moo Yi. "LF-Net: learning local features from images." In Advances in Neural Information Processing Systems, pp. 6234-6244. 2018.
[2] YI, Kwang Moo, et al. Lift: Learned invariant feature transform. In: European Conference on Computer Vision. Springer, Cham, 2016. p. 467-483.
[3] Verdie, Yannick, Kwang Yi, Pascal Fua, and Vincent Lepetit. "TILDE: a temporally invariant learned detector." In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 5279-5288. 2015.

Extent and terms of a thesis are specified in directions for its elaboration that are opened to the public on the web sites of the faculty.

Supervisor: **Ing. Tomáš Fabián, Ph.D.**

Date of issue: 01.09.2019

Date of submission: 30.04.2020

doc. Ing. Jan Platoš, Ph.D.
Head of Department

prof. Ing. Pavel Brandštetter, CSc.
Dean

I hereby declare that this master's thesis was written by myself. I have quoted all the references I have drawn upon.

Ostrava, May 15, 2020 .....

I hereby agree to the publishing of the master's thesis as per s. 26, ss. 9 of the Study and Examination Regulations for Master's Degree Programmes at VŠB – Technical University of Ostrava.

Ostrava, May 15, 2020 .....

I would like to thank my supervisor Ing. Tomáš Fabián, Ph.D. for consultations and new ideas of how to approach any encountered problems.

Abstrakt

Tato práce popisuje použití konvolučních neuronových sítí s cílem detekovat a popsat klíčové body v obrazech. Tyto obrazy pochází z datasetu, který lze považovat za složitější optoti jiným datasetům, které jsou typicky používané v této oblasti. V této práci jsou popsány dva přístupy – první používá neuronovou síť, která se skládá z částí typických pro tyto tyto algoritmy – konkrétně se jedná o detektor klíčových bodů, odhad jejich orientací a jejich popis pomocí deskriptorů. V druhém přístupu je používána neuronová síť, která běžně slouží k detekci objektů a která byla upravena tak, ať detekuje právě klíčové body.

Klíčová slova: CNN, detekce klíčových bodů, deskripce příznaků, detekce objektů, neuronové sítě

Abstract

This thesis describes work with convolutional neural networks with the aim to detect and describe keypoints in images from a non-standard dataset, which is more complex than the datasets typically used for this task. Two approaches are explored – the first one relies on a network covering the typical feature extraction pipeline, which consists of keypoint detection, orientation estimation and feature description, whereas the second approach uses an object detector network to detect and label specified keypoints.

Keywords: CNN, keypoint detection, feature description, object detection, neural networks

Contents

List of symbols and abbreviations

CNN	–	Convolutional Neural Network
DoG	–	Difference of Gaussian
FPR	–	False Positive Rate
GHH	–	Generalized Hinging Hyperplanes
NN	–	Neural Network
ROC	–	Receiver Operating Characteristic
TPR	–	True Positive Rate

List of Figures

List of Tables

List of Listings

1 Introduction

Keypoint detection and feature matching using descriptors has wide range of usage in various computer vision tasks. As such, there are many different approaches for extracting the keypoints from actual images and computing their descriptors, each having their own advantages and disadvantages. From state-of-the-art methods like SIFT [1] or SURF [2], to some more recent approaches using CNNs, there is still ongoing developement in this area.

It was already proven, that approaches using neural networks can outperform the state-of-the-art methods on typical tasks, like image stitching, where all the images are obtained from roughly the same position with only slight viewpoint changes. This work however aims to match features from images with drastically changing viewpoint. Example of such images can be seen in Figure 1, where are two rendered images of the same model. From each image, an example image patch is extracted, which corresponds to the same physical point on the model, meaning the description vectors for these patches should be similar and therefore be matched between the images.

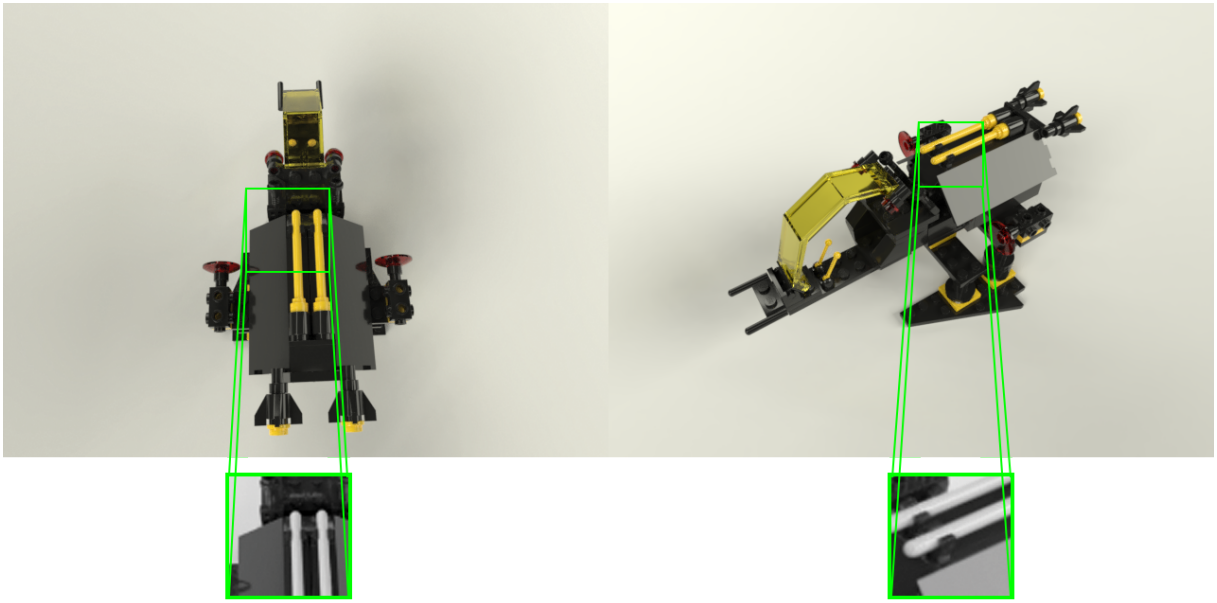


Figure 1: Example of two patches extracted from different images, which correspond to the same physical point and should be matched

In reality however, the state-of-the-art methods fail to do so and their use on images with such different viewpoints offers no usable results. The idea therefore is, that a modern approach using neural networks could be better suited for this task, since the state-of-the-art methods compute the descriptors based solely on the pixel values in the image, but the neural network could in some way learn to encode the actual physical position into the descriptors.

Specifically, an approach using a CNN based on the architecture of LIFT [3] will be explored. This network can be split into three separate parts – a keypoint detector, an orientation estimator

and a descriptor, which correspond to the steps done by the state-of-the-art methods, which, given an input image, at first detect keypoints in it, then compute their sizes and orientations and as the last step compute their descriptors. An analysis of the whole network will be provided and then a custom implementation, with some changes compared to the original architecture, which should make it better suited for the specific dataset, will be done.

Later, a more unconventional approach using an object detection network will be explored to offer an alternative way of handling the problem, since none of the feature detection networks, including the used LIFT, were created with a dataset, such as the one used in this work, in mind, meaning the achieved results may not meet the expectations. As the object detector, a YOLOv4 [4] network, which is currently considered as one of the best in this area, will be used. To suit it for this task, a single keypoint on the model can be considered an object and the network should then be able to find such selected keypoints in whole images.

2 Existing approaches

This Section will provide a brief overview of some existing methods used for feature detection and description. These can be split into two basic categories: state-of-the-art methods, which are older and more known, and approaches using neural networks, which are now gaining in popularity.

2.1 State-of-the-art methods

The first big milestone in the area of feature detection and description was the release of SIFT [1] in the year 2004. It proposed a whole pipeline on how to detect and describe keypoints in an image with scale and rotation invariance, and also a method on how to match these detected keypoints according to their descriptors. Since then, many other methods were created, with most of them being based on the original pipeline of SIFT with the aim to either achieve better results or faster computational speed.

2.1.1 SIFT

The process used in SIFT works as follows: at first, a Gaussian pyramid is created, which consists of octaves. Each octave contains several images obtained by applying a Gaussian filter on an input image, each time with different and increasing scale σ of the filter. The first octave is created from the original input image, while each subsequent one works with an image from the previous octave resized to half of its height and width. After the pyramid is completed, a Difference of Gaussian for each two neighboring images in an octave, which serves as a blob detector, is created by simply subtracting those two images. A visualization of this process can be seen in Figure 2.

Then, the DoG images are searched for local maxima and minima, by comparing the pixel values with neighboring pixels not only in a single image, but also in images with adjacent scales in an octave. This results in potential keypoint locations, however they have to be further refined and filtered. At first, the locations are interpolated from their neighborhoods to obtain more precise results and keypoints with low contrast, which means, that they were most likely created by noise, are discarded. As a second step, keypoints which are located along edges are also discarded, since they are of no significance and only keypoints located at corners should be considered.

After only significant keypoints remain, their orientations must be computed. This is done by computing gradient magnitudes and orientations in a small neighborhood around the keypoints, which are then used to create a orientation histogram with 36 bins to cover the 360 degree range around the keypoint. The resulting orientation is then calculated from the peaks in this histogram.

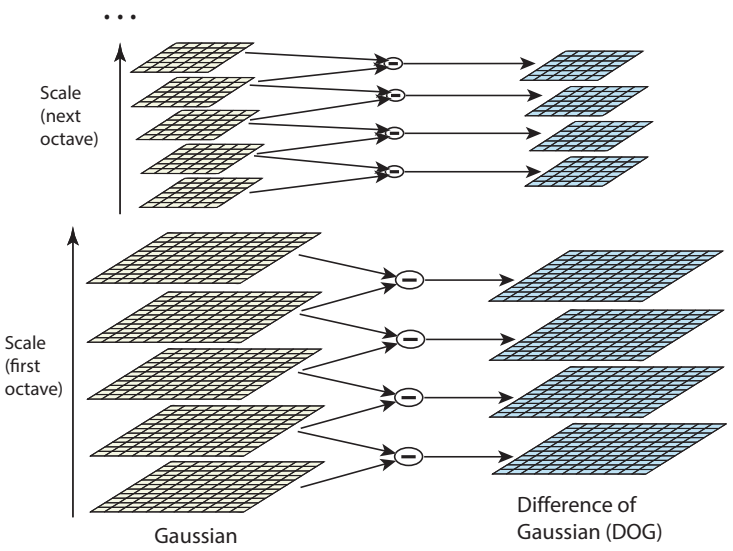


Figure 2: Illustration of a process used to obtain the DoG images in SIFT [1]

The last step is to compute a descriptor for each of the keypoints. This is done by taking a 16×16 neighborhood of each keypoint, which is divided into 4×4 blocks, resulting in a total number of 16 blocks. For each block, an orientation histogram with 8 bins is created, and the order of the bins is then sorted according to the calculated orientation of the keypoint to achieve rotation invariance. Finally, the bin values are serialized into a vector, creating is the actual descriptor, which has 128 dimensions based on the number of blocks = 16 and the number of bins = 8 in each of them.

2.1.2 Succeeding methods

Later, other methods were released. For example SURF [2] was meant to be a sped-up version of SIFT, as it used simpler box filters instead of DoG, which reduced the needed time for computation, but also influenced the results in a negative way. Both SIFT and SURF were also patented, so some other free alternatives were released as well, such as ORB [5], which aimed to be even faster, or KAZE [6], which chose different approach for keypoint detection, with the aim on better performance rather than speed.

Overall, there are many state-of-the-art methods, each offering their advantages and disadvantages and choosing the right one depends on the actual task and its requirements of either faster computational speed or better results. But even now, 16 years after its release, SIFT is still considered as a first choice for many tasks and newly developed approaches still use it as a baseline to compare results.

2.2 Methods using neural networks

In recent years, there has been an increasing number of new approaches using neural networks with the aim to outperform the state-of-the-art methods. This wasn't possible to do earlier because of hardware limitations, as even a simple network would take too much time to train, but now a sufficient technology is available and therefore the focus in many areas is shifting to neural networks and machine learning.

A lot of published works focus only on one part of the whole feature detection and description process – for example TILDE [7] introduced a keypoint detector, which could outperform the state-of-the-art methods, but works only on datasets with a fixed camera position and changing lighting and weather. Table 1, which is a reduced version of the one published in the original paper [7], shows the performance compared to some selected state-of-the-art methods. The used metric is a repeatability score, which works with pairs of input images and equals to a percentage of keypoints detected in the first image of the pair, which were also detected in the second image at corresponding locations.

Table 1: Repeatability score of TILDE compared to state-of-the-art methods on specific datasets, taken from [7]

	Webcam dataset	Oxford dataset	EF dataset
TILDE	**40.7**	**59.1**	**33.0**
SIFT	20.7	43.6	23.0
SURF	29.9	57.6	28.7
MSER	22.3	35.9	23.9

As for orientation estimation, there wasn't a lot of work dedicated to specifically this area and therefore there wasn't any big improvement compared to the process introduced in SIFT.

The opposite is true for computing the descriptors, where many new approaches were published. Some of them are different compared to the state-of-the-art methods by for example using high-dimensional descriptors (up to 4096 dimensions), compared to 128 in SIFT, or by learning a different metric to compare and match different descriptors, as in MatchNet [8], instead of using the standard L_2 distance. A more ideal approach with standard output comparable to the one in SIFT was shown for example in PN-Net [9] or DeepDesc [10], where the output descriptors have typical number of dimensions and are comparable by the L_2 distance. Table 2 contains results of both MatchNet and PN-Net compared to SIFT, with the metric being a false positive rate at 95% true positive rate on the ROC curve, where lower value corresponds to a better result.

Table 2: True positive rate at 95% false positive rate of MatchNet and PN-Net compared to SIFT, taken from [9]

	Yosemite dataset	Liberty dataset	Notre Dame dataset
PN-Net	**7.74**	8.27	**4.45**
MatchNet	8.39	**6.90**	5.76
SIFT	27.29	29.84	22.53

LIFT [3], which was chosen as the main reference for this work, focuses on all three parts of the pipeline. A custom keypoint detector is learned as well as an orientation estimator and feature descriptor, with each of these parts outperforming both the state-of-the-art methods as well as the CNN approaches shown earlier, which can be seen in Table 3.

Table 3: Matching score of LIFT compared to other CNN and state-of-the-art approaches, taken from [3]

	Strecha dataset	DTU dataset	Webcam dataset
LIFT	**37.4**	**31.7**	**19.6**
PN-Net	30.0	26.7	11.4
MatchNet	22.3	19.8	10.1
DeepDesc	29.8	25.7	11.6
SIFT	28.3	27.2	12.8
SURF	20.8	24.4	11.7

2.3 Object detectors

Compared to keypoint detectors, whose only job is to detect keypoints in an image and return their locations, object detectors have to additionally classify the detected objects as well as return their bounding boxes. This means that an object detector has to have predefined classes of objects for which it should look for in images. In first object detectors, which were created before the wide use of neural networks, this was done by creating object templates by for example using HOG [11] or even the already mentioned SIFT features. The same methods were then applied on input images and SVMs [12] were used to compare the extracted features with the predefined templates to try to classify them.

Later, new approaches using neural networks were created. Specifically in 2014, the R-CNN [13] object detector was released. This approach used convolutional neural networks and split the object detection into 2 sub-tasks – given an input image, a region proposal network was used on it to predict regions in it, which could possibly contain objects, that should be detected. These regions were then passed to a CNN, which then tried to classify them. Use of this two-step

process however made the whole detection quite slow and therefore several improved versions were later released, which aimed to increase the speed.

However in 2016, a new approach called YOLO (You Only Look Once) [14] was released. It was able to detect objects in real-time thanks to it being a single-step detector, which worked by applying a CNN on whole input image, which was then divided into grid of smaller cells. Then, predictions of bounding boxes and their probabilites were made for each cell, which were at the end weighted to obtain the final predictions. An illustration of this process can be seen in Figure 3.

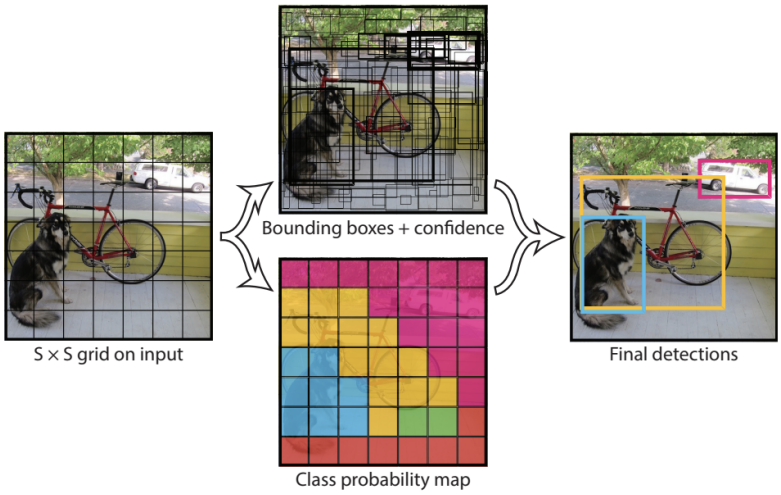


Figure 3: Process used by the YOLO network to predict bounding boxes of objects, taken from [14]

The use of a single CNN resulted in much faster computation time, which was, according to the paper, 1000 times faster than R-CNN. There were however some downsides – the predictions were not as precise and the network had problems detecting smaller objects. To handle these problems, several new versions were released, with the newest one being YOLOv4 [4]. During the development, many new features and methods were used to increase precision, such as increasing the complexity of the CNN or using residual blocks or various image augmentations. However, even after all these additions, the YOLOv4 network can still run in real-time to detect objects in videos.

3 Datasets

In this Section, some of the typically used datasets for feature detection and description will be shown. After that, the specific dataset used in this work, called the Alienator dataset, will be described to clearly show the difference compared to the standard datasets.

3.1 Standard datasets used for feature detection

The typical datasets used in most works are usually made of a large number of photos of a specific location, building or some other monument. All of these photos can be taken from a fixed position with no camera movement whatsoever, as in the Webcam dataset used in TILDE [7], with the only changing factor being the lighting and weather. Example images from this dataset can be seen in Figure 4.

Figure 4: Different images from the Webcam dataset

Other, more complex datasets can additionally contain changes in camera position and perspective. Example of this can be the Madrid Metropolis dataset from [15]. As it can be seen from Figure 5, there are some significant perspective changes, however they are still not that drastic, so even the state-of-the art methods can still have quite reliable performance on this dataset.

Figure 5: Different images from the Madrid Metropolis dataset

Both of these datasets contain real photos, which in most cases cannot be directly used as inputs and some additional processing, such as using Structure from Motion [16], must be done to obtain data usable by the specific approaches. This is however not the case in datasets from [17], which were made specifically for descriptor training and testing and already contain pre-cropped and pre-labeled image patches, so the use of these datasets is much simpler and only matter of correctly loading them. Example of such a dataset, created from photos of the Notre Dame, can be seen in Figure 6.

Figure 6: Example from the Notre Dame dataset

3.2 Alienator dataset

This dataset consists of images rendered in Blender containing a model of a ship named Alienator, which is built from LEGO bricks. The camera is aimed at the center of the model in each image, however its position always changes. In total, there are 222 different views from all around the model. Example of how different can the views be is shown in Figure 7.

Figure 7: Different views of the model from the Alienator dataset

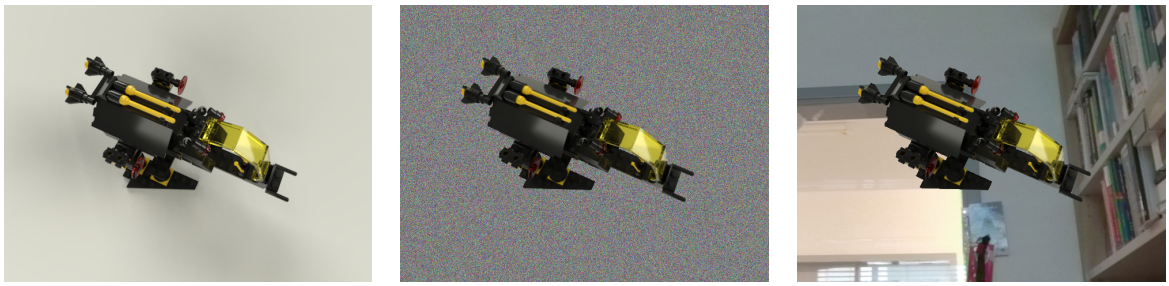
It should be clear, that a dataset like this one, with the camera moving all around the model, offers much greater challenge than the typical datasets shown before and for example the state-of-the-art SIFT [1] wasn't able to reliably and correctly match any keypoints between these different images.

Additionally, an EXR file exists for each of the color images, which contains 3D world coordinates for each pixel of the base image. Example of a colorized EXR file can be seen in Figure 8.

(a) Color image (b) EXR file containing world positions

Figure 8: Sample image from the Alienator dataset and its corresponding EXR file

With the help of the EXR files, some additional sets of images with different backgrounds were created. The EXR files were simply traversed pixel by pixel and if the world position was not a value corresponding to a position on the model, the color at that pixel was changed in the base image. The backgrounds were changed to a random noise as well as some random photos, which were obtained from a fish-eye camera. The idea is that if the background was always the same, the network could learn to detect it as a part of the keypoint, which is not desirable, and the changing backgrounds will prevent that. The different backgrounds can be seen in Figure 9. The complete dataset therefore contains 222 views of the model, with each view being used multiple times, each time with a different background.



(a) Original background (b) Noise background (c) Photo background

Figure 9: Image from the Alienator dataset with different backgrounds

3.2.1 Creating an actual dataset usable for feature detection

These base images however do not form any real dataset, which can be used to train and test the neural network. Such a dataset can be created by grouping locations in different images, which correspond to the same physical point on the model.

At first, a list of world positions/physical points on the model has to be obtained. This can be done in two ways – either by handpicking them using a simple application or by using SIFT to detect keypoints in all images, grouping them by their world positions and taking the first n

most detected. An example of keypoints obtained this way can be seen in Figure 10, where first 40 points with the most detections by SIFT are shown.

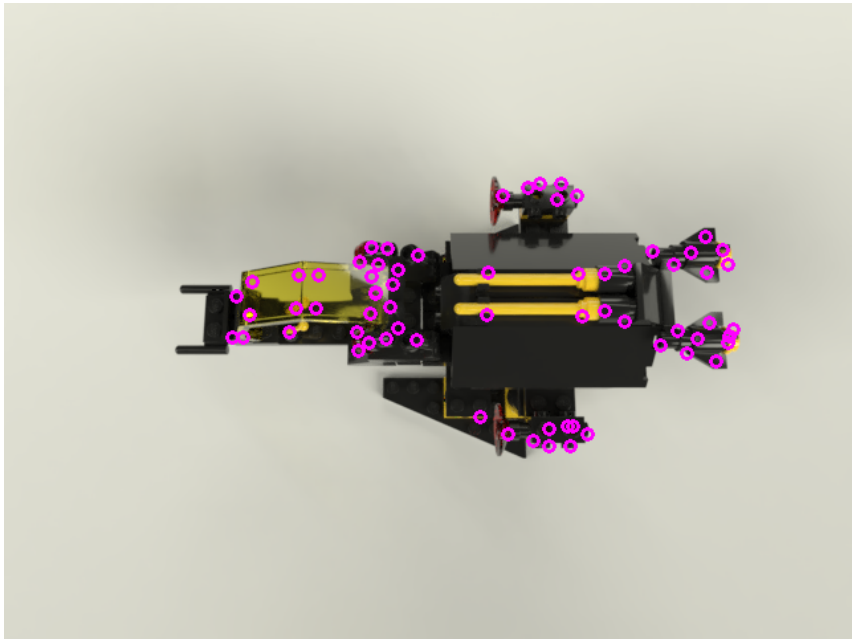


Figure 10: Most frequently detected points on the Alienator model

With the list of world positions obtained, the actual grouping and creation of the dataset can be done. Let's also consider a list of all images in the base dataset. Both of these lists can then be iterated, meaning that each image is searched for each of the physical points using the following formula:

$$x_{kp}, y_{kp} = \arg\min_{x,y} \|I_{x,y} - p\|_2 ,$$

where I is the EXR image containing 3D world positions, x, y are all the coordinates in the image, p is the 3D position of the currently searched physical point and x_{kp}, y_{kp} are the coordinates in the image, at which the distance from the point p is the lowest. This however does not automatically mean, that the image contains a keypoint corresponding to the physical point at these coordinates, because the following additional condition must be fulfilled:

$$\left\| I_{x_{kp},y_{kp}} - p \right\|_2 < t.$$

The value t is a maximum distance threshold between p and the position in the image at the coordinates x_{kp}, y_{kp}. If a position at these coordinates has a larger distance than the threshold, it is not considered a keypoint and is discarded.

4 Analysis of the LIFT network

In this Section, a the LIFT [3] neural network will be described. At first, a more general overview of the whole network will be shown, and then each of the three main parts will be described. Lastly, a run-time pipeline of the network, which uses all the trained sub-networks, will be shown, as it differs from the training architecture.

4.1 General information and architecture

As already mentioned, the whole network consists of three smaller sub-networks: the keypoint detector, the orientation estimator and the descriptor with each of them being specifically a siamese CNN.

During the training process, each of these parts is trained separately and in reverse order, meaning the descriptor is trained first, it is then used during training of the orientation estimator in its loss function and lastly, the keypoint detector is trained, which can also use the learned versions of both the descriptor and the orientation estimator to fine-tune its results. The sub-networks use some less standard methods and approaches, which are not typically used in most neural networks, so the following subsections will cover these to make the understanding of the sub-network descriptions easier.

4.1.1 Siamese network

Each of the three parts of LIFT can be considered a siamese network – a neural network with a specific architecture, which has multiple inputs, but each of them is processed by the same network with the same weights. The resulting outputs are then used in a single loss function, which combines them and calculates a loss value, which depends on all of the outputs. A simplified diagram of this architecture can be seen in Figure 11, where a siamese network for three inputs is shown.

4.1.2 Input image patches

The sub-networks use image patches as inputs during training. An image patch is a crop of a whole image with a specified size, so all the input patches have the same dimensions. Each image patch is centered at a specific point of interest, in this case specifically a keypoint in an image. How to obtain the locations was described in Section 3.2 for the Alienator dataset, however all the datasets used for keypoint detection and description have a way of obtaining these locations or are already made of pre-cropped patches.

As already mentioned in Section 3.2, each keypoint and therefore its corresponding image patch belongs to a specific class, which is a world position of a physical point on the model in the case of the Alienator dataset. With each patch having its class, specific groups of patches can be created. Specifically, pairs of patches, that either belong to the same class or are from two

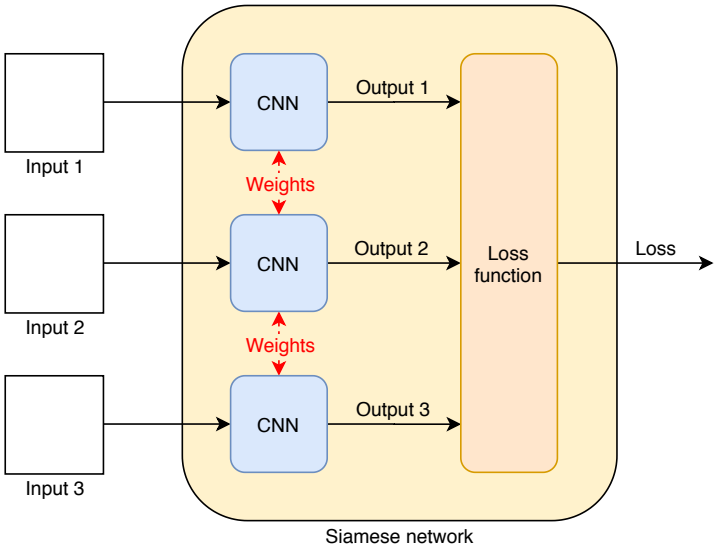


Figure 11: Simplified model of a siamese network with three inputs and shared weights

different classes, or furthermore even triplets or quadruplets with combinations of patches from same/different classes, or in the case of the quadruplets even some patches, which specifically don't contain any keypoint, can be created. An example of a triplet, which consists of two patches belonging to the same class and third from a different one, can be seen in Figure 12. These groups can then by used as inputs of the sub-networks, since they use the already described siamese architecture and therefore support multiple inputs.

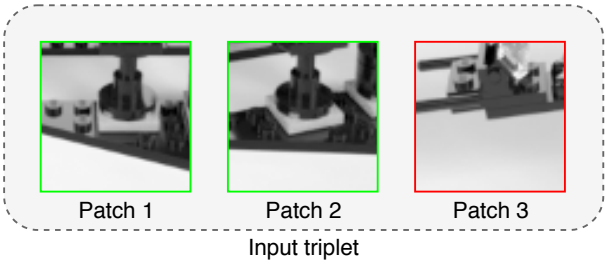


Figure 12: Example triplet of image patches

Specifically, the patches used by the LIFT network have size of 128×128 px. These are used to train the detector, while both the orientation estimator and the descriptor use their cropped versions with the size of 64×64 meaning that they contain twice as small neighborhood as the original patches.

4.1.3 Custom GHH layer

Both the detector and the orientation estimator use a non-standard GHH layer, which was firstly used in the TILDE [7] detector. It is mostly used instead of an activation function after a convolutional or a fully connected layer. It has three inputs: a vector x with D dimensions

and two parameters M and N. The first step is to iterate over the input vector and add the maximum of every M values into a new vector, resulting in a second vector with $\frac{D}{M}$ dimensions. This vector is then iterated over once again, but now the sum of each N values is added to a third and final vector, which will have $\frac{D}{MN}$ dimensions. This layer can therefore be compared to a pooling layer since it also reduces dimensions, but in the case of images it reduces their number of channels instead of their resolution.

4.1.4 Hard mining

All the sub-networks also use the process of hard mining. The assumption is that a network quite quickly reaches some level of performance and then the it stalls, because most of the inputs no longer contribute anything valuable to the learning process. The solution to this is to forward all the inputs through the network, but only use a fraction of them for backpropagation. Specifically, the inputs with the worst loss values are used and are called the hardest samples. The parameter controlling how many of the inputs are used for backpropagation is called a mining ratio and is defined as:

$$r = K_f / K_b,$$

where K_f is the batch size or the number of forwarded inputs and K_b is the number of them used for backpropagation. This allows the network to continue learning on relevant inputs.

The hard mining however takes its toll on the performance of the learning process, because ideally the number of inputs, which will be used for backpropagation, should always stay the same. This means, that the initial batch size used for the forward pass should increase along with the mining ratio. When using mining ratio of $r = 4$ or even $r = 8$, the initial batch size suddenly increases from the standard 128 to 512 or 1024. The forward pass of that many additional data and the consecutive sorting and selection of the hardest samples makes the learning much slower. However, the performance of a network learned this way is also higher.

4.2 Descriptor

The role of the descriptor is to compute a description vector for an input image patch of size 64×64 pixels. During training, a siamese architecture with three inputs is used, so the simplified model shown earlier in Figure 11 actually corresponds to the descriptor network. This also means, that triplets, which were already described in Section 4.1.2 and which contain two patches of the same class and third one corresponding to a different one, are used as inputs for training.

During run-time after the network is trained, only a single image patch is used as an input, as the siamese architecture is only useful for training and there is no reason to use multiple inputs after that, since the only goal of the network is to return a single description vector for a single input.

4.2.1 Network architecture

As already mentioned, the training architecture is a siamese network with three inputs. Each of the inputs is then processed by a CNN, which can be further split into 3 repeatable blocks, each containing a convolutional layer, a batch normalization layer, a non-linearity layer and a pooling layer. This is shown in Figure 13, whereas Figure 14 shows the whole CNN made of these blocks, which is used to compute the descriptors and therefore is used on each input in the siamese network and also later during run-time on the single image patch.

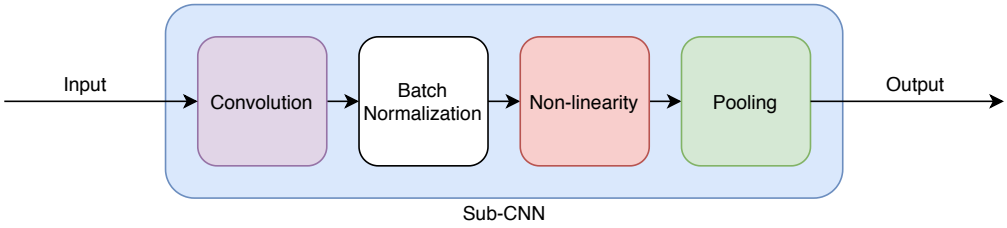


Figure 13: A repeatable convolutional block of the descriptor CNN

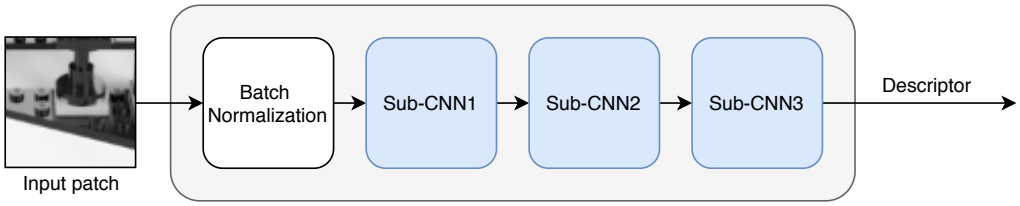


Figure 14: A CNN used to compute descriptors from image patches

The parameters of all the layers are chosen in such a way, that an input 64×64 px patch is transformed into a 128-dimensional vector and can be seen in table 4. Another possibility would be to add a single fully connected layer at the end, which would directly control the dimensionality of the output vector.

Table 4: Parameters for the convolutional blocks of the descriptor CNN

	Sub-CNN1	Sub-CNN2	Sub-CNN3
Input size	64×64×1	29×29×32	8×8×64
Number of filters	32	64	128
Filter size	7	6	5
Non-linearity	ReLU	ReLU	ReLU
Pooling type	Avg	Avg	Avg
Pooling stride/size	2	3	4
Output size	29×29×32	8×8×64	1×1×128

4.2.2 Loss function

Let's consider a single input triplet containing image patches p_1, p_2 and p_3, where p_1 and p_2 belong to the same class and are called a positive pair. p_3 is from a different class and forms negative pairs with both p_1 and p_2. The network computes descriptors for the patches and the loss value is then calculated from them with two main goals. Firstly, the L_2 distance of descriptors for the positive pair should be as small as possible. The loss value for the positive pair can therefore be defined as:

$$L_{\text{pos}}(p_1, p_2) = \|D(p_1) - D(p_2)\|_2 ,$$

where $D(p_i)$ denotes a descriptor for patch p_i.

Secondly, the L_2 distance for the descriptors of a negative pair should be larger than a specific value C called the margin, which is set to $C = 4$. This means, that any distance smaller than the margin penalizes the loss value and any distance larger than the margin results in the loss value for the negative pair being 0. There are however, as already mentioned, two negative pairs in the triplet – (p_1, p_3) and (p_2, p_3). This allows to choose the pair, that contributes more to the loss function, meaning the pair with the smaller distance. At first, let's define a formula, which chooses the smaller distance of the two negative pairs:

$$L_{\text{neg}}^{\min}(p_1, p_2, p_3) = \min(\|D(p_1) - D(p_3)\|_2 , \|D(p_2) - D(p_3)\|_2).$$

Then, the final loss value can be calculated combining the two obtained values and utilizing the margin C. An ideal result, meaning a loss value of 0, will be obtained if the distance of the positive pair is 0 and the distance of the negative pair is the value C or more. The final formula therefore looks like this:

$$L_{\text{desc}}(p_1, p_2, p_3) = L_{\text{pos}}(p_1, p_2) + \max(0, C - L_{\text{neg}}^{\min}(p_1, p_2, p_3)).$$

4.3 Orientation Estimator

The goal of the orientation estimator is to predict an angle ϕ for an input image patch of size 64×64 px. This angle is then used to rotate the patch. During training, pairs of image patches belonging to the same class are used, meaning the network has a siamese architecture with two inputs. It is also necessary to already have a trained version of the detector, since it is used in the loss function of this network. During run-time, similarly to the descriptor network, only a single image patch is used as an input, for which the network returns its orientation.

4.3.1 Network architecture

A single CNN, which processes both input patches in the siamese network during training as well as the single input during run-time is also used here. It once again uses the repeatable

convolutional blocks shown earlier in Figure 13 with the addition of blocks shown in Figure, which contain a fully connected layer, a batch normalization and a GHH layer. The whole CNN made from these blocks is then shown in Figure 16, while the parameters for the convolutional and the fully connected blocks are available in Tables 5 and 6 respectively. It is also worth noting, that the output from the last fully connected block contains two values, which are handled a sine and cosine of the angle of the input patch. They are therefore passed to a `atan2` function, which returns the actual value of the angle.

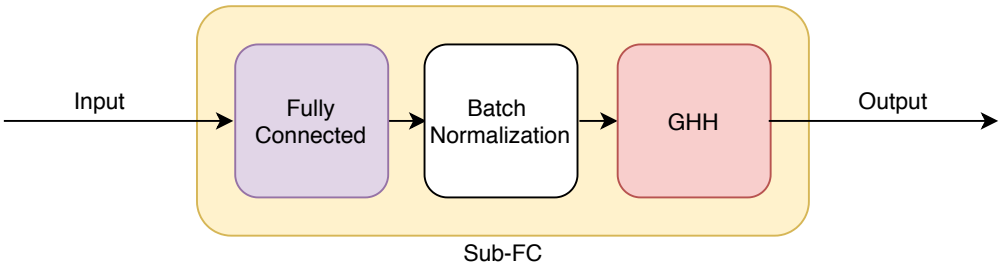


Figure 15: A repeatable fully connected block of the orientation estimator CNN

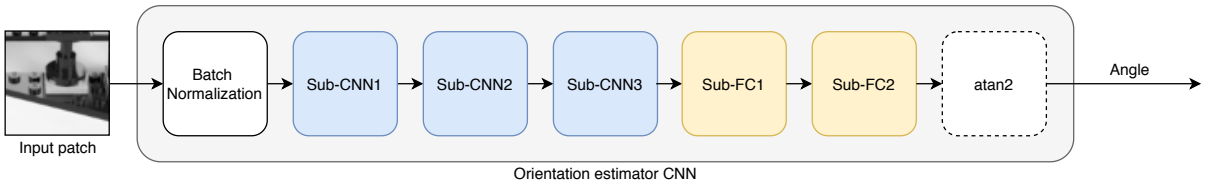


Figure 16: A CNN used to obtain orientations for image patches

Table 5: Parameters for the convolutional blocks of the orientation estimator CNN

	Sub-CNN1	Sub-CNN2	Sub-CNN3
Input size	64×64×1	29×29×32	8×8×64
Number of filters	32	64	128
Filter size	7	6	5
Non-linearity	ReLU	ReLU	ReLU
Pooling type	Avg	Avg	Avg
Pooling stride/size	2	3	4
Output size	29×29×32	8×8×64	1×1×128

4.3.2 Loss function

As already mentioned, the orientation estimator network uses the trained version of the descriptor in its loss function. Also, the non-cropped, 128×128 versions of the input patches P_1 and

Table 6: Parameters for the fully connected blocks of the orientation estimator CNN

	Sub-FC1	Sub-FC2
Number of neurons	1600	32
GHH parameters	$N = 4, M = 4$	$N = 4, M = 4$
Output dimensions	100	2

P_2 along with the coordinates of keypoints in them x_1 and x_2 are required. Then, angles ϕ_1 and ϕ_2 can be obtained by applying the orientation estimator CNN on the input 64×64 patches p_1 and p_2. With this done, the loss function can be defined as:

$$L_{\text{ori}}(P_1, x_1, \phi_1, P_2, x_2, \phi_2) = \|D(RC(P_1, x_1, \phi_1)) - D(RC(P_2, x_2, \phi_2))\|_2 ,$$

where $RC(P_i, x_i, \phi_i)$ is a function, which rotates the patch P_i by an angle ϕ_i around a center x_i and then crops the patch to a size of 64×64 px still using x_i as the center and finally $D(p_i)$ is a description vector for a patch p_i returned by the trained descriptor network. The aim therefore is to find such angles for the input patches, for which the distance of descriptors of their rotated versions is the smallest.

4.4 Detector

The detector network returns a scoremap for an input image, which is then used to find location(s) of keypoints in it. There is however a significant difference compared to the descriptor and orientation estimator networks – whereas these networks always work with image patches of a specified size, the detector network uses image patches only for training. During run-time, the detector is applied on a whole image of any size. This means, that the methods used to obtain keypoint locations from the scoremaps are different, as the image patches for training are made in such a way, that they contain only a single keypoint in their center, but the whole images used during run-time can contain a large number of keypoints.

During training, a siamese architecture is still used. The network has four inputs and therefore uses quadruplets of patches, where the first three are the same as in the triplets used by the descriptor network, whereas the fourth patch specifically contains no keypoint. Also, the patches here have the size of 128×128 px and therefore the neighborhood of the keypoint is twice as big compared to the 64×64 patches used in the other sub-networks. During run-time, as with the previous sub-networks, only a single input is used, since using the siamese architecture makes no sense.

4.4.1 Network architecture

Both the patches during training and the images during run-time are once again processed by the same CNN. The possibility to use it on images of different sizes comes from the fact, that the only layer with learnable weights is a convolutional layer. If the network contained any fully connected layer, this would not be possible, as any fully connected layer requires a constant output size from previous layer, which is not the case with inputs of different sizes.

The actual CNN is a fairly simple network consisting of only 3 layers – a batch normalization layer used to normalize the input image, a convolutional layer and finally the GHH layer, which was described earlier in Section 4.1.3. An illustration of this CNN can be seen in Figure 17, whereas the actual parameters for the layers are shown in Table 7.

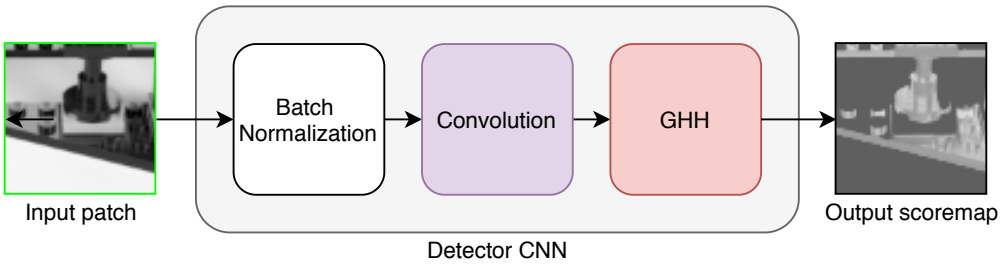


Figure 17: A CNN used by the detector network to obtain a scoremap

Table 7: Layer parameters for the single branch CNN of the descriptor network

	Descriptor CNN
Number of filters	16
Filter size	25
GHH parameters	$N = 4, M = 4$

4.4.2 Loss function

Let's consider the quadruplet of patches P_1, P_2, P_3 and P_4, which is used as an input during training. In it, P_1 and P_2 belong to the same class, P_3 belongs to a different one and P_4 contains no keypoint. Then, the detector CNN can be applied on each of them to obtain their scoremaps S_1, S_2, S_3 and S_4. Then, a softargmax function, which computes a center of mass location in such scoremap can be defined as such:

$$\text{softargmax}(S) = \frac{\sum_y \exp(\beta S(y)) y}{\sum_y \exp(\beta S(y))},$$

where y are all locations in S and $\beta = 10$ is a parameter controlling the smoothness of the function. Then, the first part of the loss function can be defined as:

$$L_{\text{pair}}(P_1, P_2) = 1 - \frac{p_1 \cap p_2}{p_1 \cup p_2},$$

where p_i is a 64×64 patch created by cropping the original 128×128 patch P_i at a location x_i, which is obtained from the softargmax function. Intersection and union in this case refer to areas of the patches p_i depending on their locations x_i in the original patches P_i. For example, if both x_1 and x_2 were the same, it would mean, that areas of the smaller patches overlap exactly and their intersection and union would be the same, resulting in the loss value of 0. On the other hand, if x_1 was for example in the upper left corner of the original patch and x_2 was in the bottom right, they would not overlap at all, their intersection would be 0 and the loss value would therefore be 1. This uses the fact, that the input patches are always centered at a keypoint, which means, that ideally both the locations obtained by the softargmax function from the scoremaps should also be at the center, which would once result in the loss value being 0. Now, the second part of the final loss function can be defined and looks like this:

$$L_{\text{class}}(P_1, P_2, P_3, P_4) = \sum_{i=1}^{4} \alpha_i \max(0, (1 - \text{softmax}(S_i) y_i))^2,$$

where softmax is a log-mean-exponential soft maximum function, which returns a single score value and $y_i = -1$ and $\alpha_i = \frac{3}{6}$ if $i = 4$, and $y_i = 1$ and $\alpha_i = \frac{1}{6}$ otherwise. The aim of this loss function is to have the scores of patches P_1, P_2 and P_3, which contain a keypoint, as high as possible whereas the score of the patch P_4, which contains no keypoint, should be as low as possible. With both loss values calculated, they can simply be summed to obtain the total loss value. The final loss function therefore looks like this:

$$L_{\text{det}}(P_1, P_2, P_3, P_4) = L_{\text{class}}(P_1, P_2, P_3, P_4) + L_{\text{pair}}(P_1, P_2).$$

Later, after the network was already trained, the L_{pair} function can be replaced by another version to fine-tune the network and possibly increase its performance. This version requires to have both the orientation estimator and the descriptor networks to already be trained and looks like this:

$$\tilde{L}_{\text{pair}}(P_1, P_2) = \left\| D(RC(P_1, x_1, O(p_1))) - D(RC(P_2, x_2, O(p_2))) \right\|_2,$$

where $O(p_i)$ is the angle returned by the orientation estimator, $RC(P_i, x_i, \phi_i)$ rotates the patch P_i by an angle ϕ_i around a center x_i and then crops it to a size of 64×64 px and $D(p_i)$ is the description vector returned by the descriptor network. This allows the detector to possibly find better keypoint locations in the original patches, as the firstly used L_{pair} returned 0, when the predicted locations x_i were in the centers of the patches, but now, the loss value directly depends on the distance of their descriptors, which can be better for some other locations.

4.5 Run-time architecture

As already mentioned, the run-time architecture of the network differs from the training one. Firstly, the siamese architectures of the sub-networks are only useful for training and have no use, after all the networks are trained. This means, that during run-time, only the actual CNNs, which work with single inputs, can be used.

The second change is in usage of the detector. It was trained using patches, but during run-time it works with whole images. Even then, it only returns a scoremap, so some additional processing must be done. Specifically, the input image is resized several times similarly to SIFT [1], but instead of applying Gaussian filters on those images, they are passed to the detector to get their scoremaps. Then, a non-maximum suppression is used on these scoremaps, which results in keypoint locations in the original image. Patches from these locations can then be created by cropping the original image and those patches can then be simply passed to the orientation estimator and descriptor networks respectively, to obtain a descriptor for each of them. The resulting keypoint locations and descriptors can then be used by standard feature matching algorithms used by the state-of-the-art methods. An illustration of the whole pipeline can be seen in Figure 18.

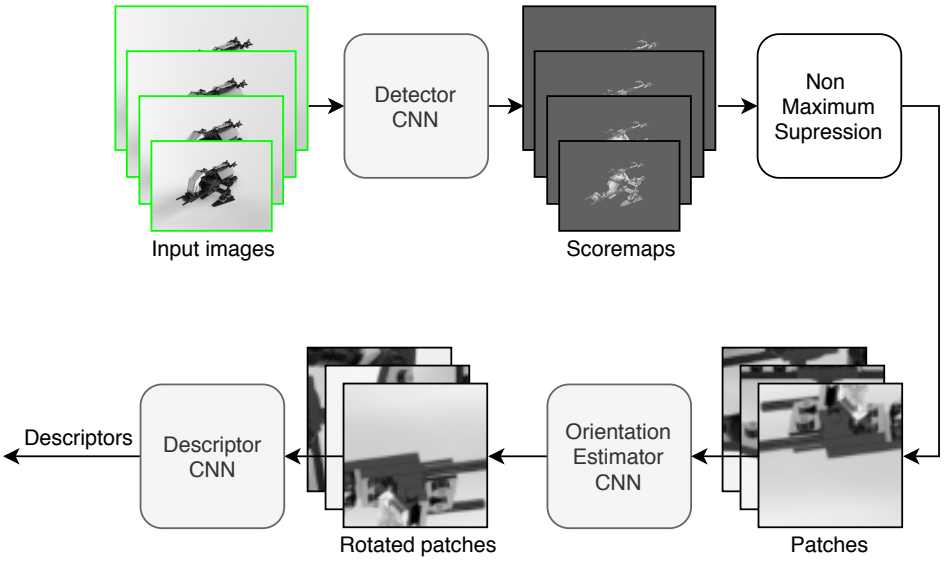


Figure 18: Run-time architecture of the whole LIFT network

5 Custom implementation of LIFT

This Section will cover some of the implementation details of the LIFT network and describe changes made to the original architecture to better fit the Alienator dataset. Also, some information related to the dataset, such as its format and loading, will be described. Then, the implementation of the sub-networks and the network as a whole will be shown.

5.1 General information and changes in architecture

The initial creation of the dataset, as generally described in Section 3.2, was done in a C++ application with the help of the OpenCV library. The rest of the implementation, which means creating and training actual network was done in Python using the Tensorflow library and its Keras API.

The original architecture of LIFT described in Section 4 was also changed to better fit the Alienator dataset. Specifically, it was decided to omit the orientation estimator, because its purpose in typical feature detection tasks is to return an angle, which is used to rotate the image patch in a way, that matching patches look as similar as possible and therefore are rotated the same way. However in the Alienator dataset, a simple 2D rotation of the patches would not achieve much, because the different camera locations make it impossible to rotate all patches corresponding to the same keypoint in such a way, that they all look similar. It was therefore decided, that the orientation estimator would not provide any significant benefit in this specific scenario and the description vector returned by the descriptor should be viewed as an encoded physical point on the Alienator model rather than a more general description of the content of the image patches.

5.2 Dataset format

In Section 3.2, a general information about how to obtain an usable dataset from the initial images was shown. This was implemented in C++ and a specific output format was chosen for this implementation of the LIFT network. Specifically, two files are created, one containing training and the other one testing data. Each row in these files contains following information:

```
class_id, file_name, x, y, width, height
```

where the row describes an image patch in file `file_name` with dimensions `width` and `height`, which contains keypoint of class `class_id` centered at position [`x,y`] in the image. The files also contain specific rows with `class_id` is set to -1, which correspond to patches, that contain no keypoint. These patches are used to train the detector and therefore must also be contained in the dataset. The text files therefore contain all necessary information to load all the described patches containing keypoints and assign them their corresponding classes.

5.2.1 Loading the dataset

To load the dataset, the lines in the dataset text file are read one by one and for each of them, the corresponding image is loaded and cropped at the specified location to create the image patch of the required size. Each patch is then also assigned a label according to its class. Specifically, a method called `datasets.alienator` exists and takes a parameter corresponding to either the training or the testing dataset file. The method returns three arrays: `patches, labels` and `nonkp_patches`, which can then be further processed.

5.2.2 The `LiftDataset` class

A specific class was created to be used as the dataset wrapper for the LIFT network, as Tensorflow and Keras only work with and have existing classes prepared for datasets, in which an epoch can be simply defined. This is not the case for this network, because input triplets and quadruplets are generated from the image patches during run-time and the number of such combinations is much higher, than a number of inputs in a single epoch of a typical dataset. Because of this, the `LIFTDataset` class was created to serve the purpose of generating new inputs during training of the networks.

The class takes two mandatory parameters in the constructor, `patches` and `labels`, along with one optional parameter – `nonkp_patches`. Considering the loaded Alienator dataset, two instances of this class can be created – first for the training data and second for the testing. Such dataset format of four arrays, excluding the ones containing non-keypoint patches, is also used by some other datasets, for example the built-in MNIST dataset in Tensorflow. The class was therefore made to be able to process any dataset in this format, as it simply generates random noise images to use as the non-keypoint patches, if the optional parameter was not passed.

The class then constructs an internal dictionary, where the unique labels/class ids are used as keys and each of them has a corresponding array of patches, which belong to the class specified by it. Also, each patch is resized to the size of 128×128 px, as it is the largest size used by LIFT, specifically by the detector. Such dictionary then allows to simply get triplets and quadruplets of patches, which are required as inputs for the sub-networks.

Specifically, the class has methods `get_det_quadruplet`, which returns a single quadruplet of randomly selected patches to be used by the detector, where the first two patches belong to the same class, the third one to a different one and the fourth one is a patch not containing any keypoint. Then, the method `get_det_quadruplets`, which has a parameter `batch_size`, returns an array or a batch containing this specified number of quadruplets. Similarly, the methods `get_desc_triplet` and `get_desc_triplets` work with triplets of patches, which are used by the descriptor network. Additionally, patches returned by these methods are of the size of 64×64 px and are cropped from the original 128×128 px patches at the center, as the descriptor uses these smaller patches. The class is therefore ready to be used by both the descriptor and detector networks.

5.3 Descriptor

As already mentioned, the Tensorflow library with its Keras API was used to implement the network. The API makes the task of creating simple networks very easy, but also contains easily understandable function and classes, which can be used to build more complex and non-standard networks. A base building block of a neural network in Keras is a model object. Specifically, two versions are available – a more simple sequential model, which can be created by creating an instance of the `Sequential` class, or a more complex one, which is called simply `Model` and uses a functional API. To create the descriptor network with a siamese architecture, both of these models were used and a class simply called `Descriptor` was created to encompass all its required behavior.

5.3.1 Sequential model and the CNN for a single input

Let's at first focus on the sequential model. After it is created, different layers can be added to it by calling its method `add`, which takes an instance of a `Layer` class as a parameter. Such layers can be obtained from `keras.layers`, where the implementations of commonly used layers, such as convolutional, fully connected, pooling and many others, are located. Using this, a CNN, which was described in Section 4.2.1 and is used on a single image patch to compute its descriptor, can be created. A example of how to create it with the specific parameters shown in Table 4, can be seen in Listing 1, which contains an implementation of method `get_branch_model` of the class `Descriptor`, that simply creates the model, adds the necessary layers to it and then returns it to be used later. It is also worth noting, that the first layer in the Sequential model must be created with the parameter `input_shape`, so the network can correctly generate weights, which depend on the size of the input.

```
def get_branch_model(self):
    model = K.Sequential()
    model.add(K.layers.BatchNormalization(input_shape=(64, 64, 1)))

    model.add(K.layers.Conv2D(filters=32, kernel_size=7))
    model.add(K.layers.BatchNormalization())
    model.add(K.layers.Activation(K.activations.relu))
    model.add(K.layers.AveragePooling2D(pool_size=2, strides=2))

    model.add(K.layers.Conv2D(filters=64, kernel_size=6))
    model.add(K.layers.BatchNormalization())
    model.add(K.layers.Activation(K.activations.relu))
    model.add(K.layers.AveragePooling2D(pool_size=3, strides=3))

    model.add(K.layers.Conv2D(filters=128, kernel_size=5))
    model.add(K.layers.BatchNormalization())
    model.add(K.layers.Activation(K.activations.relu))
    model.add(K.layers.AveragePooling2D(pool_size=4, strides=4))
```

```
    model.add(K.layers.Flatten())

    return model
```

Listing 1: Creating a single descriptor CNN using the `Sequential` model from Keras

After the model is created, it can be used to obtain descriptors for input image patches by calling its method `predict` or `predict_on_batch`. Using it directly after the creation would however yield no actual results, as it must firstly be trained. To do that, another model will be created, this time using the functional API, which will correspond to the siamese architecture necessary to train the descriptor network.

5.3.2 Functional API and the siamese architecture

The first step, instead of directly creating an instance of a model, as in the case of the Sequential model, is to define inputs. This is done by creating instances of the `Input` class and specifying their input shapes. This allows, compared to the Sequential model, to create multiple inputs instead of a single one, which is tied to the first layer added to the model.

With the inputs created, layers can be once again added. However instead of adding them to a existing model, the created `Layer` objects can be called as functions, which take as a parameter either one of the created inputs or an output of another layer, which was also used this way. Also, instead of a single layer, a whole model can be used in the same way. This can be seen in the beginning of Listing 2, where at first a single instance of the CNN described earlier in Section 5.3.1 is created. Then, three inputs are created and the CNN is applied on each of them to get their descriptors. This achieves the desired behavior of a siamese network, since all the inputs are processed by the same network with the same weights.

```
def __init__(self, learning_rate=0.001):
    self.branch_model = self.get_branch_model()

    self.p1 = K.Input((64, 64, 1))
    self.p2 = K.Input((64, 64, 1))
    self.p3 = K.Input((64, 64, 1))

    self.d1 = self.branch_model(self.p1)
    self.d2 = self.branch_model(self.p2)
    self.d3 = self.branch_model(self.p3)

    self.output_loss = custom_losses.descriptor_triplet_loss()([self.d1, self.d2, self.d3])

    self.siamese_model = K.Model(inputs=[self.p1, self.p2, self.p3], outputs=self.output_loss)
    self.siamese_model.compile(optimizer=K.optimizers.Adam(learning_rate),
    ↪  loss=K.losses.mean_absolute_error)
```

Listing 2: Creating the siamese descriptor network using the functional API of Keras

As a next step, the three resulting descriptors must be combined in a loss function, which was described in Section 4.2.2, to obtain a single loss value. Usually, when a model created using the functional API has multiple outputs, a loss function is applied on each of them separately. This is however not the desired behavior here, so to overcome this issue, the loss function was created as another layer with the help of the `Lambda` class in Keras, which allows to use any implementation of a custom behavior as a layer in the models. Listing 3 shows such implementation, where the function `descriptor_triplet_loss` returns an instance `Lambda` class, which takes another function as a parameter. This passed function must also accept a single parameter, which corresponds to the input, that is passed to that layer in a network. Usage of this layer was already shown in Listing 2, where it was called on the three descriptors obtained from the inputs, which were simply concatenated to a list, which serves as the single input of this layer.

```
def descriptor_triplet_loss(margin=4.0):
    def calculate_loss(outputs):
        loss_pos = K.backend.sqrt(K.backend.sum(K.backend.square(outputs[0] - outputs[1]),
        ↪  axis=1, keepdims=True))

        dist_1_3 = K.backend.sqrt(K.backend.sum(K.backend.square(outputs[0] - outputs[2]),
        ↪  axis=1, keepdims=True))
        dist_2_3 = K.backend.sqrt(K.backend.sum(K.backend.square(outputs[1] - outputs[2]),
        ↪  axis=1, keepdims=True))
        d_neg = K.backend.minimum(dist_1_3, dist_2_3)
        loss_neg = K.backend.relu(margin - d_neg)

        return loss_pos + loss_neg

    return K.layers.Lambda(calculate_loss)
```

Listing 3: Using a `Lambda` layer to implement the descriptor loss function

Last two rows in Listing 2 then show the actual creation of the model. At first, an instance of the `Model` class is created, which has two parameters – `inputs`, which simply contains the inputs created in the beginning, and `outputs`, which corresponds to the last layer(s) of the model. As each added layer was called as a function on an input or a result from previous layer, the created model has a clearly path of how to get from its `inputs` to its `outputs`. The last step is to call the `compile` method of the model, which is necessary for the model to be able to be trained. Two main parameters are the `optimizer` and `loss`, which tell the model, which optimizer and loss function to use during training. With this done, the model is ready to be trained.

5.3.3 Training the model with hard mining

A model in Keras is usually trained by calling its `fit` method. That is however not possible in this case because of two reasons – firstly, the method takes either an array or a custom `Dataset`

object from Tensorflow as an input data and works with epochs. The dataset used here is however contained in a custom class, as described in Section 5.2.2, where an epoch is not clearly defined. Secondly, by using the `fit` method, the process of hard mining, which was described in Section 4.1.4, is not possible.

Instead, a method `train_on_batch` of the model is used and the remaining logic is handled by the `Descriptor` class. Specifically, a method `hardmine_train`, whose implementation can be seen in listing 4, exists. The parameter `inputs` is a list, which contains three arrays, which correspond to the three inputs of the network and `mining_ratio` controls the actual hard mining process. If it is set to 1, the network is simply trained on all the inputs. With any higher value, the actual process of hard mining is done. Firstly, the method `predict_on_batch` of the model is called, which returns an array of loss values for the input triplets, as the last layer in the model is the actual loss function. Then, the `argsort` function is used on the array to get indices of the highest loss values, which are then used to filter the inputs. The model is then trained using those filtered inputs by calling the method `train_on_batch`. The method however requires a second parameter, which is an array of expected results for the inputs, but since the last layer is already a loss function, a basic array of zeros is passed, since the loss values should ideally be as close to zero as possible.

```
def hardmine_train(self, inputs, mining_ratio):
    if mining_ratio > 1:
        losses = self.siamese_model.predict_on_batch(inputs).numpy().flatten()
        train_with = int(losses.size / mining_ratio)
        training_indices = np.argsort(losses)[::-1][:train_with]
        inputs[0] = inputs[0][training_indices]
        inputs[1] = inputs[1][training_indices]
        inputs[2] = inputs[2][training_indices]

    return self.siamese_model.train_on_batch(inputs, np.zeros(inputs[0].shape[0]))
```

Listing 4: Implementation of a method used for training with hard mining

The overlaying logic controlling the value of mining ratio and how many iterations should be used is then implemented in the wrapper class, which will be described later in Section 5.5. After the network is trained, the Sequential model of the CNN can then be used to obtain the actual descriptors for input patches, as the siamese model uses a reference of it and therefore its weights are also updated during training. Additionally, the `Descriptor` class has methods `save_weights` and `load_weights` to be able to save and restore the weights of the trained network.

5.4 Detector

Once again, a class simply called `Detector` was created and contains all the necessary functionality of the detector network. All the principles used in the implementation of the descriptor

network, which were described in Section 5.3, are also used by this class with necessary changes made to correspond to the detector network architecture described in Section 4.4.

Specifically, the Sequential model corresponding to the single CNN contains the specific layers described in Section 4.4.1. Also, another important change was made to it – whereas the first layer in the model for the descriptor CNN, as shown in Listing 1, had the parameter `input_shape` set to `(64, 64, 1)`, as it always takes 64×64 px patches as input, the CNN used here has it set to `(None, None, 1)`. This allows it to take images of any size as an input, as long as they are grayscale, which is necessary during run-time, when full images are used instead of patches. It is possible to do so, because the only layer with learnable weights is a convolutional layer, which directly does not depend on the input shape compared to for example fully connected layers, with whom this would not be possible.

The siamese model was also changed to have four inputs instead of three. These inputs can also once again have set their `input_shape` set to a fixed values, in this case `(128, 128, 1)`, since the siamese model is used only for training, during which patches of this size are always used. They are however then resized by the network to a size of 48×48 px, which was originally done in the official implementation[1] of LIFT, possibly to make the learning faster. Another `Lambda` layer was created for this, which simply uses the function `image.resize` from base Tensorflow, which resizes all images in a batch to a specified size. This layer is therefore applied on each input and after that, they are processed by the created CNN to obtain their scoremaps. Then, the `softargmax` function, which was described in Section 4.4.2 and is implemented as another `Lambda` layer, is applied on these scoremaps resulting in three output values for each – x, y coordinates and a score.

After this step, another change compared to the descriptor network was made. It was already mentioned in Section 4.4.2 that the detector network can use different loss functions for initial training and fine-tuning. To achieve this behavior, two models were created which differ only in the last `Lambda` layer that represents the actual loss function. Both of these models are compiled and then a simple boolean value passed to the `hardmine_train` method can determine, which model will be used for training, since both of them will update the same weights in the Sequential model. With this, the class is complete and can be used similarly as the `Descriptor` class to train the network and obtain scoremaps of input images after it is done training.

5.5 The `Lift` wrapper class

To encompass both the `Detector` and `Descriptor` classes as well as to add additional functionality, another class simply called `Lift` was created. It has one mandatory parameter in the constructor – `model_dir`, which specifies the folder, which will be used to save and load the files containing weights of the trained networks. Another two optional parameters can be passed – `train_dataset` and `test_dataset`, which should be instances of the `LiftDataset`

[1] `https://github.com/cvlab-epfl/tf-lift`

class described in Section 5.2.2. These are optional because the networks can already be learned and the class only used for obtaining keypoints and detectors from images, in which case there is no reason to have any dataset loaded, as it would only take space in memory and would not be used.

The class then has methods `train_descriptor` and `train_detector` to train the corresponding networks and both of these methods have the same parameters – `iterations` to specify, for how many iterations to train (where one iteration means training on one batch), `batch_size` to specify the number of inputs in each batch and two parameters controlling the hard mining – `mine_ratio_start` and `mine_ratio_end`. The network then start training with `mine_ratio_start`, which is doubled, until it equals `mine_ratio_end` and the total number of iterations is split in such a way, that each mining ratio is used for the same amount of iterations.

Finally, methods `get_descriptors` and `get_scoremaps` exist to obtain descriptors for image patches and scoremaps for input images by passing them through the trained networks. These methods can be used separately as well as by another method – `get_keypoints_descriptors`, which takes a single input image, detects keypoints in it by using the detector network and non-maximum suppression and finally obtains descriptors for these keypoints by using the descriptor network. The method then returns two lists containing the keypoints and their descriptors respectively, where the keypoints are using the class `KeyPoint` of the OpenCV library, so they can be directly used by it. Unfortunately, this process is quite slow and can take several seconds for a single image because of the used non-maximum suppression, in which the image is scaled and passed through the network several and then the keypoints are located by iterating over the resulting scoremap.

6 Experiments

In this Section, the results of the implemented LIFT [3] network will be evaluated. Then, the focus will be shifted to the YOLOv4 [4] network to describe, how it can be trained and used to have similar functionality as a feature detector. After that, the results of such trained network will also be shown.

6.1 LIFT

As each part of the network was trained separately, they will also be evaluated on their own, starting with the descriptor, which was trained first. There were some less successful attempts in the beginning, so they will be also covered to show, what had to be changed to obtain better results.

6.1.1 Descriptor

At first, the descriptor was trained on the Notre Dame dataset shown earlier in Figure 6 to provide a baseline of how should the trained network work, since the dataset was made specifically for this task and was used by many approaches to compare results.

To present the results in an understandable and informative way, they will be visualized. Let's once again consider input triplets, which were used during training, but now from the testing dataset. For each patch in the triplet, its descriptor can be obtained by the network. Then, similarly as in the loss function described in Section 4.2.2, a L_2 distance for the positive pair can be calculated as well as for worse of the two negative pairs resulting in two distances for the triplet. This can be done for many other triplets generated from the testing dataset to obtain two lists of positive and negative distances, which can then be used to create their histograms. Such histograms for the Notre dame dataset can be seen in Figure 19, where the positive distances are shown in blue and the negative ones in orange.

Ideally, the two histograms should not overlap at all, which would mean that some threshold value t could be selected as such, that every distance below it would correctly correspond to a positive pair and any distance above it to a negative pair. In real cases however, the histograms will almost always overlap, so the goal is to have the overlapping as little as possible, which is the case for the Notre Dame dataset and would be, in ideal situation, the case even for every dataset.

In the Alienator dataset, the base images were split into two parts – training and testing. Then, to make training and evaluation faster, several points were handpicked on the model from whose the datasets were created, as described in Section 3.2. This means that the training dataset contains patches from the training set of images, whereas the testing patches contain the same keypoints as the training ones, but from a different set of images.

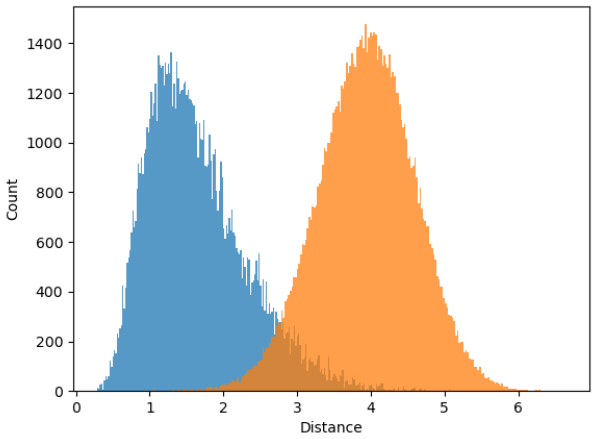


Figure 19: Histograms of descriptor distances for the Notre Dame dataset

At first, the keypoints were selected in such a way, that the symmetric versions of them would be considered as two different classes, meaning that there was left and right version of each point. However, after training the network with this dataset and once again visualizing the resulting histograms, they were overlapping much more. The histograms are shown in Figure 20 and an unusual peak can be seen at the beginning of the histogram of negative distances. This most likely resulted from the mentioned symmetric points, since the network works with cropped image patches without larger neighborhood and on this scale, the left and right versions of each object look very similar.

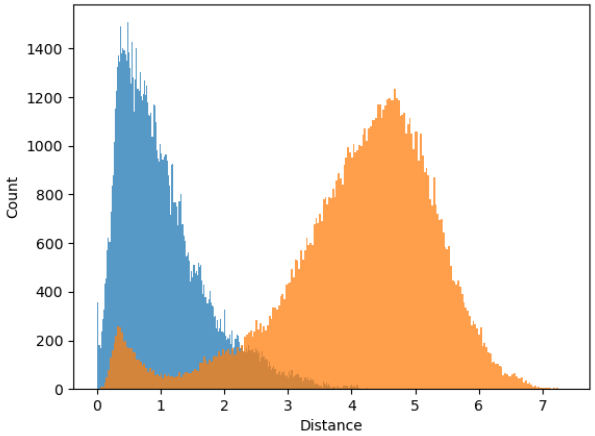


Figure 20: Histograms of descriptor distances for the Alienator dataset with separated symmetric keypoints

As a next step, a different dataset was created in which the symmetric points are grouped and considered the same one. The network was once again trained and the resulting histograms

can be seen in Figure 21. Now, there is no noticeable peak at the beginning of the negative histogram, which means, that classifying the left and right versions of the points separately was the cause of this behavior in the previous case. Now however, there will be no distinction between the left and right versions when working with whole images, however this is something, that is not the goal of feature detectors. The overlap area of the histograms is also bigger, than in the case of the Notre Dame dataset, however this is to be expected, since the Alienator dataset is more complex.

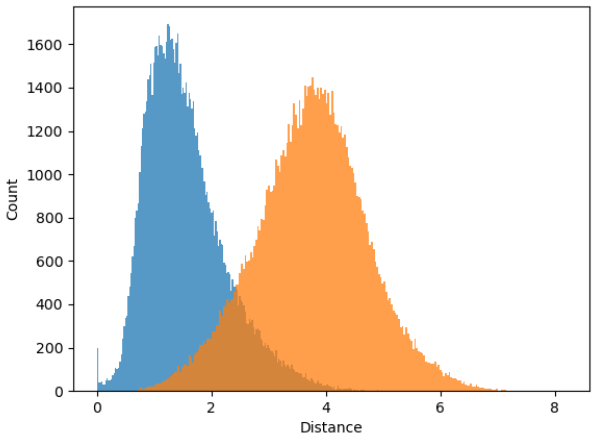


Figure 21: Histograms of descriptor distances for the Alienator dataset with grouped symmetric keypoints

Finally, a last version of the dataset was created by once again using the approach of grouping the symmetric keypoints, but instead of handpicking the points on the model, SIFT was used to detect keypoints in all the dataset images, these were then grouped according to their physical positions on the model and 40 most detected points were chosen. The network was trained one last time on this dataset to see, how the results compare to the handpicked points. Figure 22 once again shows the histograms, which are now overlapping even more, which can be attributed to the higher complexity of the dataset, as it contains more classes.

To offer numerical results, a ROC curve was created. To do so, the resulting descriptor distances can be split into two parts by choosing a threshold value t. Then, any distance below this value is considered a distance of matching descriptors, and any above it should therefore correspond to non-matching ones. Then, the following values can be obtained:

- true positives – number of positive pair distances correctly classified as positive,
- false negatives – number of positive pair distances wrongly classified as negative,
- false positives – number of negative pair distances wrongly classified as positive,
- true negatives – number of negative pair distances correctly classified as negative.

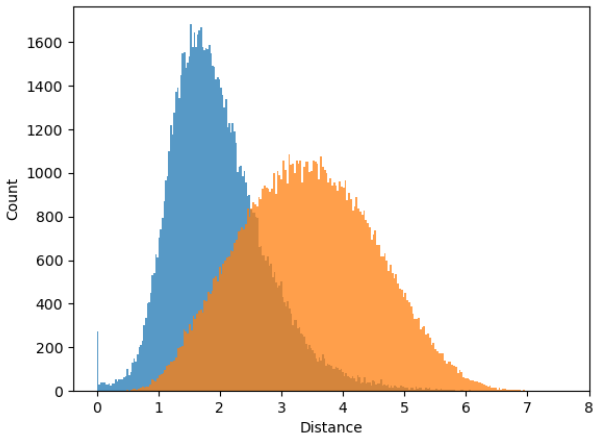


Figure 22: Histograms of descriptor distances for the final version of the Alienator dataset

From these values, true positive rate and false positive rate can be calculated as such:

$$TPR = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}},$$

$$FPR = \frac{\text{false positives}}{\text{false positives} + \text{true negatives}}.$$

This is done for different values of the threshold t ranging between the smallest and the largest distance and then, the ROC curve is created from these values by using the false positive rate as the x axis and the true positive rate as y. These ROC curves for the different versions of the Alienator dataset, as well as the Notre Dame dataset can be seen in Figure 23. Additionally, SIFT was also used on the final version of the Alienator dataset to show, how it compares to the learned network. To get a numerical result, a value of true positive rate at 95% false positive rate is commonly used, so Table 8 contains these values.

Table 8: Descriptor results for different datasets

	FPR @ 95% TPR
LIFT - Notre Dame	6.2%
LIFT - Alienator with separate symmetric keypoints	9.5%
LIFT - Alienator with grouped symmetric keypoints	18.5%
LIFT - Final version of Alienator with 40 keypoints	49.2%
SIFT - Final version of Alienator with 40 keypoints	98.3%

Interestingly enough, the numerical result for the Alienator dataset with non-grouped symmetrical points seems better, however when looking at the actual ROC curves, its curve sways

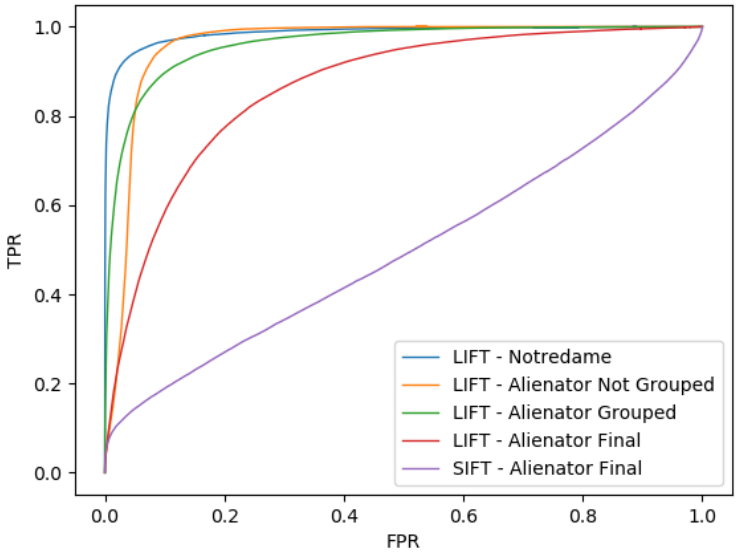


Figure 23: ROC Curves for different datasets

away from the y axis earlier, than for the dataset with grouped points. This is the result of the histograms overlapping earlier, as seen in Figure 20, which means that less results can be classified as true positives before some incorrect false positives appear. Also, the results for the final version of the dataset are significantly worse than for the version with less keypoints which were handpicked, even though they use the same approach of grouping symmetric points. This is most likely the result of increased size of the dataset as well as the chosen keypoints, since the handpicked ones were selected in such a way, that they were fairly easily distinguishable, but the keypoints with the largest number of detections by SIFT may not have this property.

6.1.2 Detector

The detector network was trained next, using both the Notre Dame and the Alienator datasets to provide a comparison between them. After the training was done, pairs of images were selected on which the non-maximum suppression, which was described in Section 4.5, was used to obtain keypoint locations in them. Then, patches were cropped at these locations and passed to the trained descriptor networks to obtain their descriptors. Finally, a standard keypoint matcher implemented in OpenCV was used to match these keypoints and their descriptors between the images. Figure 24 show such matches between two photos of the Notre Dame, for which the trained detector and descriptor were used. Most of these matches in the image seem correct, but unfortunately, no numerical results can be provided for this dataset since there has to be a known mapping between the images to evaluate the precision.

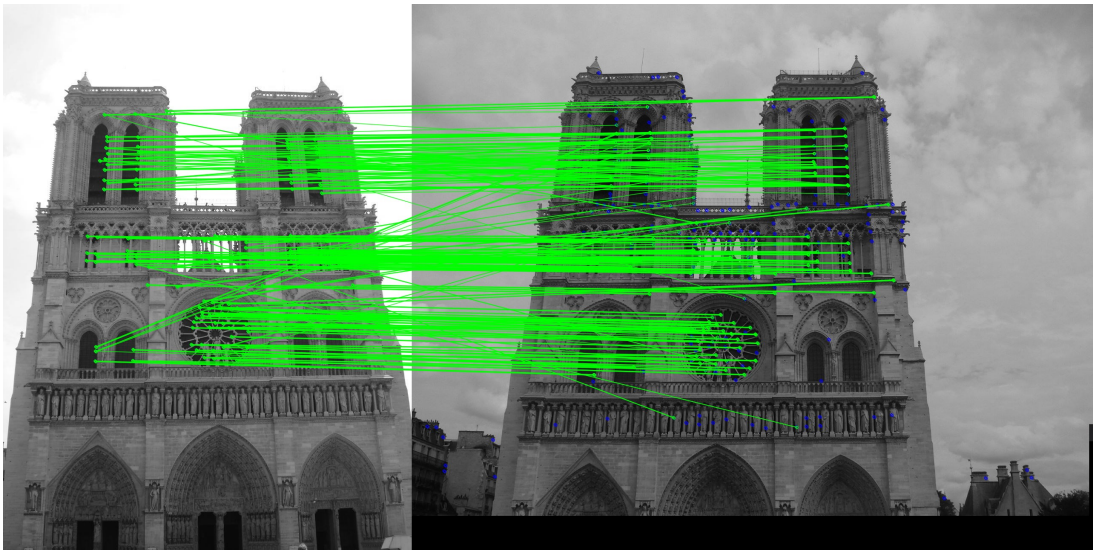


Figure 24: Detected and matched keypoints on two photos of the Notre Dame

For the Alienator dataset however, some results can be computed with the help of the EXR files. Specifically, each match refers to two keypoints – one in the first image and another one in the second image. The physical locations at these keypoints can be obtained from the EXR files and then compared to evaluate, if the match is correct. Then, a simple ratio of correct to all matches can be used to give an idea about performance of the whole pipeline. Such results were obtained by selecting ten random pairs of images from the testing set, counting the number of detected keypoints and matches and finally computing the average ratio. This was done with the trained network as well as with SIFT to compare their results, which can be seen in table 9.

Table 9: Results of the trained LIFT network compared to SIFT

	LIFT	SIFT
Number of detected keypoints	996	4058
Number of all keypoint matches	996	138
Number of correct matches	61	16
Percentage of correct matches	6.1%	11.5%

Unfortunately, the results do not seem promising as the the percentage of correct matches for SIFT is higher. Another thing worth noting is that for the LIFT network the number of detected keypoints and matches is the same, but for SIFT the number of matches is significantly lower. This is caused by a ratio test used by SIFT, where for each keypoint in the first image, two potential matches are obtained. Then, the descriptor distances for these matches are compared and if they are close to each other, both matches for the keypoint are discarded. This simple process filters large number of matches, which are potentially wrong. However it was confirmed

by the authors of LIFT, that this ratio test should not be used on the results returned by the network, as it also filters correct matches, which was verified.

The results shown are average for 10 pairs of randomly selected images from the testing dataset. When however selecting the images manually and examining the results, it becomes clear that SIFT detects a large number of correct matches in images, where the camera position changed only slightly, which is to be expected. However in the case of LIFT, the number of correct matches does not change that much. Exameple of this behavior can be seen in Figure 25, where are two images with similar camera positions matched by SIFT. Only the correct matches corresponding to approximately the same position on the model are drawn and the total number of these is 53 for SIFT, whereas only 16 correct matches were found by LIFT in the same images. The opposite can be seen in Figure 26, where the model is viewed from vastly different positions. The matches drawn were obtained from LIFT with total number of 3 matches, whereas SIFT did not detect a single correct match.

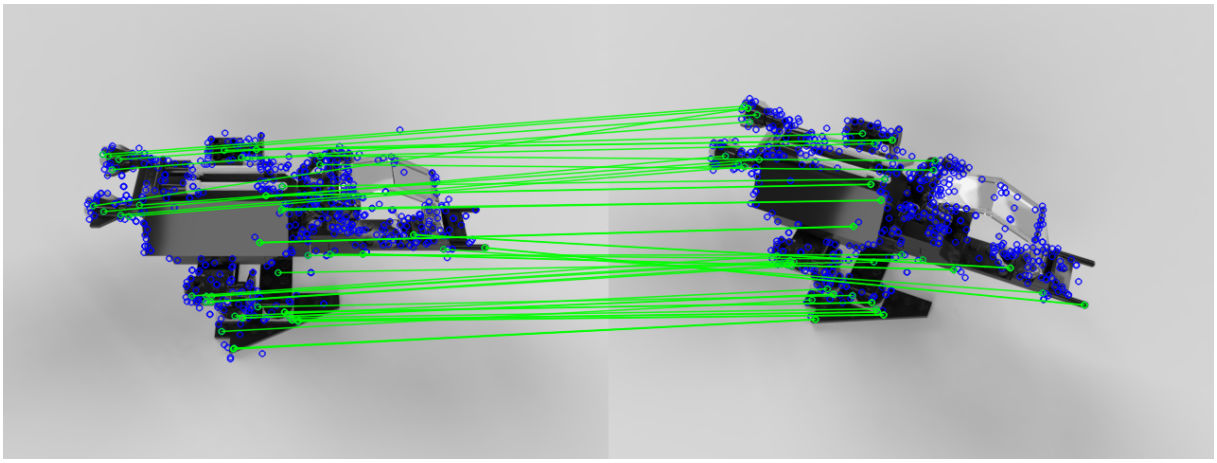


Figure 25: Correct matches obtained by using SIFT on images with similar viewpoints

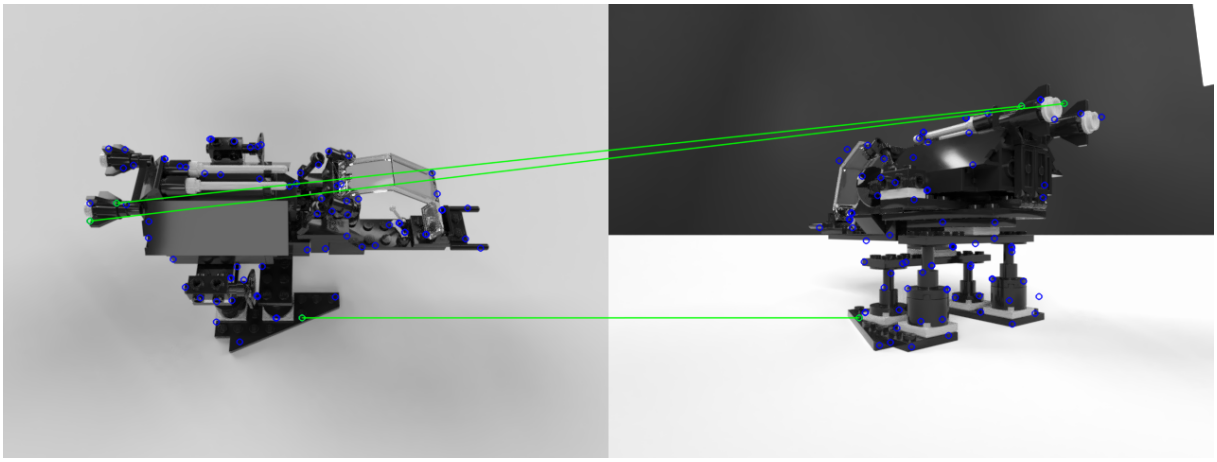


Figure 26: Correct matches obtained by using LIFT on images with different viewpoints

This however does not change the fact, that the ratio of correct matches to all matches is lower for the LIFT network. SIFT has a large advantage of being able to filter the matches bz=y using the ratio test, which cannot be done for LIFT. The total number of correct matches may be in some cases higher for LIFT, but the filtering of these correct matches was only possible because of the EXR files, which are not available for normal pictures and photos, so using it on these types of images does not seem very beneficial and a different approach may be more useful.

6.2 YOLOv4

As already mentioned, the YOLOv4 network is mainly an object detector. However when looking at a single object from a close distance, as it is the case in the Alienator dataset, various features and keypoints on it can also be considered objects, since the pictures contain enough details. The biggest difference compared to typical feature detectors is that object detectors can only detect the specific objects, on which they were trained, whereas keypoint detectors are able to find different new keypoints in each image. However with the Alienator dataset, the main goal is to detect keypoints on that specific model and these keypoints will always look the same and therefore the usage of an object detector also seems like a valid approach. This also means that no descriptors have to be computed as there is no reason for feature matching, because the detector already predicts the labels or classes of the detected keypoints.

As the YOLOv4 network is already implemented in the C++ framework Darknet [18] and available on the author's GitHub [2], the only necessary task is to create a dataset in a specific format used by it and edit specific config files to set up the network. After that, it can be trained and evaluated using the methods available in the framework.

6.2.1 Dataset format

The format used by the Darknet framework slightly differs from the one used earlier, which was described in Section 5.2. Instead of a single text file, which contains all the information, each image in the training or testing dataset has its own text file with the same name. Each row in these files then contains information about single object in the corresponding image and has the following format:

```
object_class x_center y_center width height
```

`object_class` is the same identifier as `class_id` in the LIFT dataset. The remaining values also correspond to the format used previously, however instead of simply using absolute positions and dimensions in pixels, the values used here are relative to the image dimensions, so if for example the image had width of 400 px and the x coordinate of an object was 40, the entry `x_center` would have to be 0.1. After all the needed images are labeled in their corresponding

[2] `https://github.com/AlexeyAB/darknet`

text files, another file has to be created, which can be called simply `train.txt`. This file has to contain a path to every image, which should be used for training and all these images have to have the corresponding text files created earlier. Another file in the same format can be created for the testing/validation dataset.

Finally, a file, which was in this case called simply `alienator.data`, has to be created to summarize all information about the dataset. Content of this file can be seen in Listing 5, where `classes` is the total number of unique object classes in the dataset, `train` and `valid` are files describing the training and validation datasets, which were described above, `names` is a path to another file, which contains a name for each class in the dataset and finally `backup` is a path to a folder, where the network weights will be saved during training.

```
classes = 12
train  = alienator/train.txt
valid  = alienator/test.txt
names = alienator/alienator.names
backup = alienator/backup/
```

Listing 5: Using a `Lambda` layer to implement the descriptor loss function

With all this done, the dataset is prepared and the only thing left before the network can be trained is to edit a configuration file, which contains the actual architecture of the network, which will be used. Specifically, the dataset used here was similar to the first version used to train the LIFT descriptor, in which the left and right versions of each symmetric point are considered separate keypoints.

6.2.2 Configuring and training the network

In Darknet, each network is described by a single configuration file, which contains all its layers and their parameters. In the case of the YOLOv4 network, a base file called `yolov4-custom.cfg` exists, which has to be modified according to the used dataset. The recommended changes are once again available on the author's GitHub and basically consist of changing the total number of iterations and the number of filters in several layers depending on the number of classes in the dataset, changing the batch size and number of subdivisions, so the training data can fit into the memory of the used GPU and possibly increasing the input size, which can increase the accuracy at the cost of slower training. One additional thing was added in the case of the Alienator dataset – a new row with `flip=0`, which tells the network not to flip the images during training, which is needed, if the dataset contains symmetric objects, which was the case here.

After all of this is done, training of the network can begin. This is simply done by running the executable file of the compiled Darknet and passing it several parameters – the `*.data` file containing information about the dataset, the config file, which was modified earlier and a file containing pre-trained weights of the network, which is available for download. Specifically, the command used to train the network on the Alienator dataset looks as follows:

```
darknet detector train alienator/alienator.data alienator/yolov4-alienator.cfg
↪       yolov4.conv.137
```

During the course of training, a chart, which contains loss values, is periodically updated and can be used as an indicator, if the network is learning correctly. Such chart, which resulted from training the network on the Alienator dataset, can be seen in Figure 27. The network also saves its weights into a file every 1000 iterations, so the training can be continued by using these files, if an unexpected crash occurs.

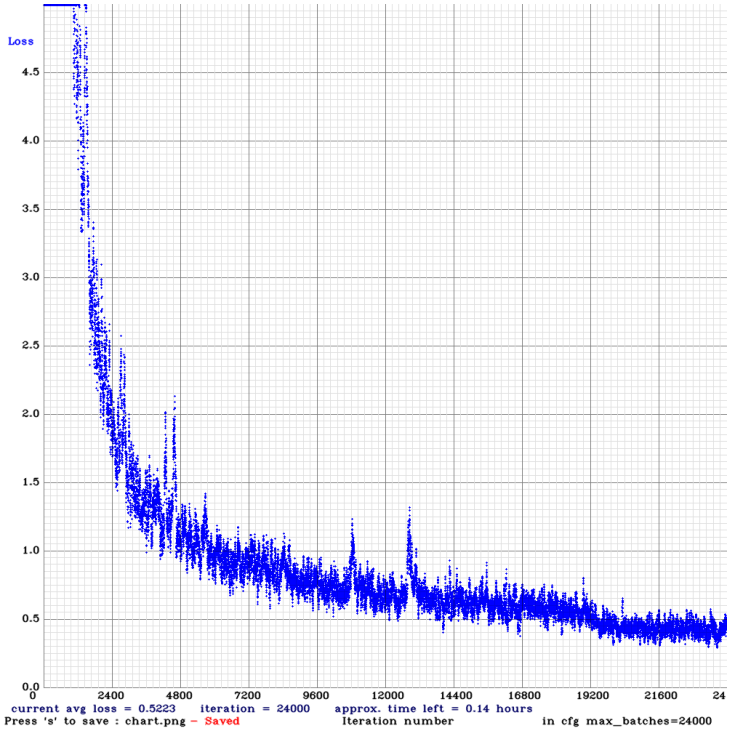


Figure 27: Loss chart from training the YOLOv4 network on the Alienator dataset

6.2.3 Results

The dataset, on which the network was trained, contained 12 unique keypoints, which were handpicked using the C++ application briefly mentioned in Section 5.2, which also allows to export the dataset into the format used by Darknet, that was described earlier in Section 6.2.1. More specifically, the dataset contains symmetrical versions of keypoints, meaning that there are left and right versions of 6 objects on the model. Using this dataset with an input size of 416×416 times and 24000 iterations, the training took over 30 hours on a RTX 2070 Super GPU. The required training time can therefore be considered a downside of the YOLOv4 network.

Similarly as with the LIFT network, the base dataset images were split into training and testing part. Additionally, five real photos of the built model were downloaded from the internet and labeled manually to see, how will the network work on them, since only the rendered

images were used during training. The Darknet framework has already implemented methods for evaluation, which can be simply run, if the `*.data` file shown in listing 5 contains the row `valid`, which refers to the testing part of the dataset. Then, a corresponding command can be run, which returns several values used for evaluation. This was done for both the testing part of the rendered images as well as the photos and the results can be seen in Table 10.

Table 10: Results of the trained YOLOv4 network

	Testing dataset	Real photos
True positives	298	22
False positives	23	1
False negatives	20	17
Precision	0.93	0.96
Recall	0.94	0.56
IoU	85.86%	71.13%

Looking at the results, they seem much better than in the case of the LIFT network. In both cases, the precision, which denotes how many of the detected objects were classified correctly, is over 90%, which is the result of small number of false positives. However in the case of the real photos, the number of false negatives, which negatively affects the value of recall, is also quite high. This may however be caused by the chosen photos. Figure 28 shows the detected keypoints in two of these photos, where in the first one almost all keypoints were detected, whereas in the second one, only two were detected. When looking at the photos, a clear difference can be seen as the first photo is in higher quality than the second one and the angle, at which was the second photo was taken, also looks a bit unusual. This means that the quality and type of photos used definitely affects the results.

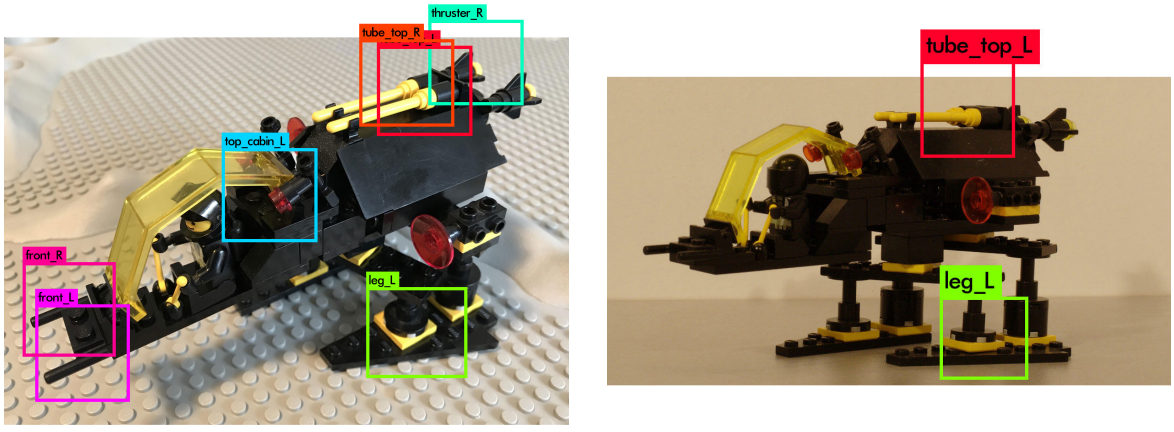


Figure 28: Detected keypoints by the YOLOv4 network in real photos

Lower value of recall, which denotes how many of the objects, which should be detected, were actually detected, is however not as big of a problem compared to a situation, where the value of precision would be low, because it is more desirable to not detect anything at all than to detect many objects incorrectly. The last mentioned value – Intersection-over-Union refers to the bounding boxes and is denotes as an area of overlap of the labeled and predicted bounding boxes over the area of their union and ideally they should overlap exactly resulting in IoU of 100%. The lower IoU for the real photos is mostly result of them being labeled by hand, which is not as precise as the generated data of the Alienator dataset.

7 Conclusion

The goal of this thesis was to implement an existing approach for feature detection and description which uses convolutional neural networks and use it on a non-standard dataset, on which the state-of-the-art methods do not provide very satisfying results. Specifically, the used Alienator dataset consists of rendered images of a model from different positions around it, which is more complex compared to the traditional datsets used for this task, where the camera position does not change as much. In this dataset, the keypoints which should be matched correspond to the same physical point on the model and the idea was that the network could somehow encode these physical positions into the descriptors rather than just describe the pixel values.

The selected approach was the LIFT [3] network, which was analyzed and then implemented using the Tensorflow library and its Keras API. Some changes were made to the original architecture with a goal to make it more suited for the Alienator dataset. Specifically, the orientation estimator was omitted and some other simplifications were made compared to the original network. Even with these modifications, the resulting network failed to outperform the state-of-the-art SIFT [1] on the Alienator dataset. Results of the descriptor part of the network looked promising and outperformed SIFT using the standard metric for comparing descriptors, however when paired with the detector and used on whole images, the results were not nearly as good as expected. The network was able to match a small number of keypoints between specific pairs of images with big difference in camera position in which SIFT was not able to make any matches, however the average percentage of correct matches was still several times lower than the one of SIFT.

Considering the unsatisfying results, an alternative approach using the object detection network YOLOv4 [4] was proposed and explored. To create a dataset used by this network, a set of keypoints was selected on the model and each of them was considered a different object. The network was trained on this dataset and its results were noticeably better, than those of the LIFT network. There however is a significant difference between these two approaches – whereas typical keypoint detectors should be able to detect new keypoints in different images, an object detector only detect those objects/keypoints, on which it was trained. This however does is not a problem in this specific case, because the main goal was to detect keypoints solely on the Alienator model. Also, the network is much faster than LIFT, since there is no need for descriptor matching as the detected keypoints are already labeled. Whereas in LIFT the processing of a single image can take several seconds, the YOLOv4 network can ever run in real-time on videos if a sufficient GPU is used.

Overall, it can be said that the YOLOv4 network is more suited for this specific task as it clearly provides better results. The number of different keypoints which the network detects can also be easily increased by simply creating a bigger dataset with the only downside being the time needed to train the network, as it took over 30 hours to train it for 10 keypoints on a higher-end GPU.

References

1. LOWE, David G. Distinctive Image Features from Scale-Invariant Keypoints. *Int. J. Comput. Vision*. 2004-11, vol. 60, no. 2, pp. 91–110. ISSN 0920-5691. Available from DOI: 10.1023/B:VISI.0000029664.99615.94.

2. BAY, Herbert; ESS, Andreas; TUYTELAARS, Tinne; VAN GOOL, Luc. Speeded-Up Robust Features (SURF). *Comput. Vis. Image Underst.* 2008-06, vol. 110, no. 3, pp. 346–359. ISSN 1077-3142. Available from DOI: 10.1016/j.cviu.2007.09.014.

3. YI, Kwang Moo; TRULLS FORTUNY, Eduard; LEPETIT, Vincent; FUA, Pascal. LIFT: Learned Invariant Feature Transform. *Computer Vision - Eccv 2016, Pt Vi*. 2016, vol. 9910, pp. 17. 467–483. Available from DOI: 10.1007/978-3-319-46466-4_28.

4. BOCHKOVSKIY, Alexey; WANG, Chien-Yao; LIAO, Hong-Yuan Mark. YOLOv4: Optimal Speed and Accuracy of Object Detection. *arXiv e-prints*. 2020-04, arXiv:2004.10934. Available from arXiv: 2004.10934 [cs.CV].

5. RUBLEE, Ethan; RABAUD, Vincent; KONOLIGE, Kurt; BRADSKI, Gary. ORB: An Efficient Alternative to SIFT or SURF. In: *Proceedings of the 2011 International Conference on Computer Vision*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 2564–2571. ICCV '11. ISBN 978-1-4577-1101-5. Available from DOI: 10.1109/ICCV.2011.6126544.

6. ALCANTARILLA, Pablo Fern$#225;ndez; BARTOLI, Adrien; DAVISON, Andrew J. KAZE Features. In: *Proceedings of the 12th European Conference on Computer Vision - Volume Part VI*. Florence, Italy: Springer-Verlag, 2012, pp. 214–227. ECCV'12. ISBN 978-3-642-33782-6. Available from DOI: 10.1007/978-3-642-33783-3_16.

7. VERDIE, Yannick; YI, Kwang Moo; FUA, Pascal; LEPETIT, Vincent. TILDE: A Temporally Invariant Learned DEtector. *Proceedings of the Computer Vision and Pattern Recognition*. 2015. Available also from: http://infoscience.epfl.ch/record/206786.

8. XUFENG HAN; LEUNG, T.; JIA, Y.; SUKTHANKAR, R.; BERG, A. C. MatchNet: Unifying feature and metric learning for patch-based matching. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015-06, pp. 3279–3286. ISSN 1063-6919. Available from DOI: 10.1109/CVPR.2015.7298948.

9. BALNTAS, Vassileios; JOHNS, Edward; TANG, Lilian; MIKOLAJCZYK, Krystian. *PN-Net: Conjoined Triple Deep Network for Learning Local Image Descriptors*. 2016. Available from eprint: arXiv:1601.05030.

10. SIMO-SERRA, E.; TRULLS, E.; FERRAZ, L.; KOKKINOS, I.; FUA, P.; MORENO-NOGUER, F. Discriminative Learning of Deep Convolutional Feature Point Descriptors. In: *2015 IEEE International Conference on Computer Vision (ICCV)*. 2015-12, pp. 118–126. ISSN 2380-7504. Available from DOI: 10.1109/ICCV.2015.22.

11. DALAL, Navneet; TRIGGS, Bill. Histograms of Oriented Gradients for Human Detection. In: *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1 - Volume 01*. USA: IEEE Computer Society, 2005, pp. 886–893. CVPR '05. ISBN 0769523722. Available from DOI: 10.1109/CVPR.2005.177.

12. HEARST, Marti A. Support Vector Machines. *IEEE Intelligent Systems*. 1998-07, vol. 13, no. 4, pp. 18–28. ISSN 1541-1672. Available from DOI: 10.1109/5254.708428.

13. GIRSHICK, Ross; DONAHUE, Jeff; DARRELL, Trevor; MALIK, Jitendra. Rich feature hierarchies for accurate object detection and semantic segmentation. *arXiv e-prints*. 2013-11, arXiv:1311.2524. Available from arXiv: 1311.2524 [cs.CV].

14. REDMON, Joseph; DIVVALA, Santosh; GIRSHICK, Ross; FARHADI, Ali. You Only Look Once: Unified, Real-Time Object Detection. *arXiv e-prints*. 2015-06, arXiv:1506.02640. Available from arXiv: 1506.02640 [cs.CV].

15. WILSON, Kyle; SNAVELY, Noah. Robust Global Translations with 1DSfM. In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2014.

16. WU, C. Towards Linear-Time Incremental Structure from Motion. In: *2013 International Conference on 3D Vision - 3DV 2013*. 2013, pp. 127–134.

17. BROWN, M.; HUA, G.; WINDER, S. Discriminative Learning of Local Image Descriptors. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 2011-01, vol. 33, no. 1, pp. 43–57. ISSN 0162-8828. Available from DOI: 10.1109/TPAMI.2010.54.

18. REDMON, Joseph. *Darknet: Open Source Neural Networks in C* [http://pjreddie.com/darknet/]. 2013–2016.